Integration of OpenModelica in Ptolemy II

Mana Mirzaei Lena Buffoni Peter Fritzson
Department of Computer and Information Science (IDA),
Linköping University, Division SE-581 83, Linköping, Sweden

Abstract

In this paper we present the work done to integrate OpenModelica into the Ptolemy II framework for modeling large-scale concurrent systems. To this end a dedicated computational model for OpenModelica has been defined in Ptolemy II, and support for tool-interaction has been implemented. This implementation will allow to simulate existing Modelica models by the OpenModelica compiler in a heterogeneous context together with models from other computational domains.

Modelica, Ptolemy II, hierarchical system modeling, concurrent systems

1 Introduction

Distributed, concurrent systems are becoming increasingly common. However, they are complex to develop. Therefore a large number of computational models and tools for modeling and developing such systems has emerged. The Ptolemy project aims to support such heterogeneous modeling in Ptolemy II[1], an open-source software framework for modeling, simulation and design of large concurrent real-time systems. This framework is a system- level design environment that provides the possibility of combining several variants of models of computation (MoCs) in one hierarchical heterogeneous model. Ptolemy II also supports an actor-oriented view of a system where the basic building blocks of a system are concurrent components called actors which communicate through messages sent via interconnected ports.

Ptolemy II currently supports numerous concurrent programming models, such as process networks, discrete event or continuous time models and interfaces with other simulations tools such as Matlab [2]. In this paper we present the integration of Modelica models in Ptolemy II through an integration with the OpenModelica tool.

Modelica[3] is a non-proprietary, object-oriented, equation based language aimed at modeling complex multi-domain physical systems. Moreover, this language is supported by a number of free and commercial tools, in particular by OpenModelica [4], an open source compiler and tool suite, complete with a text and graphical modeling editor (OMEdit) for modeling and simulation of cyber-physical systems.

The Modelica Standard Library contains a large number of models and functions from multiple domains. Integrating Modelica in Ptolemy would enable the use of these models in concurrent system modeling, as well as models from many other existing libraries.

In this paper we present the integration process of OpenModelica into the Ptolemy II framework, and illustrate use of Modelica models from within Ptolemy II on a simple example.

The paper is structured as follows, Section 2 presents the basic blocks of the integration between OpenModelica and Ptolemy, Section 3 discusses the simulation mechanism, Section 4 presents the performances of the proposed implementation, Section 5 highlights some related works and finally Section 6 briefly sums up the contents of the paper.

2 Integration Architecture

Ptolemy II is composed by a number of different domains and supports an actor-oriented modeling paradigm. Actors are the main building blocks of the system. They are concurrent components that communicate through interfaces called ports. Relations define the interconnection between these ports, and the communication structure between actors. Viewing a system as a structure of actors emphasizes its causal structure and its concurrent activities, along with their communication and data dependencies. A consequence of an actor-oriented view of a system is the decoupling of the transmission of the data from the transfer of control [5].

2.1 OpenModelica director

Ptolemy II is modular and relies on a well-organized package structure where the core packages support the data model, or abstract syntax, of Ptolemy II design. These packages also support the abstract semantics that allows domains to interoperate with maximum information abstraction. The user interface packages provide support for our XML file format, called MoML, together with a visual interface for constructing models graphically. The library packages provide actor libraries that are domain polymorphic, meaning that they can operate in a variety of domains. Additionally, the domain packages provide domains, each of which implements a model of computation. Some of which provide their own, domain- specific actor libraries.

A model of computation (MoC) is defined as a set of rules that governs the interactions between components and determines the semantics of a model. Moreover, these rules determine when actors perform internal computation and update their internal state. The semantics of the computation model are implemented through the concept of *Director*.

In order to integrate OpenModelica into Ptolemy II, it is necessary to create a dedicated computation domain with the corresponding director. Since Modelica is a language designed for continuous and discrete event modeling modeling of physical systems and variables described using DAEs, the continuous-time domain in Ptolemy II which models physical processes and supports mixtures of discrete and continuous behaviors is considered the most suitable for the Modelica language. Therefore the OpenModelica domain extends the continuos time domain already present in Ptolemy. Figure 1 shows the OpenModelica director with its parameters, such as the number of iterations or the solver to be used .

The integration work described in this paper involves three sub-packages of the domain package: `kernel`, `demo` and `lib`. The kernel package provides the software architecture for the Ptolemy II data model, or abstract syntax. This abstract syntax has the structure of clustered graphs. The classes in this package support entities with ports, and relations that connect the ports.

2.2 OpenModelica actor

Actors, the basic building blocks of a system, are the executable entities used to build the models. The OpenModelica actor reads one or more inputs from

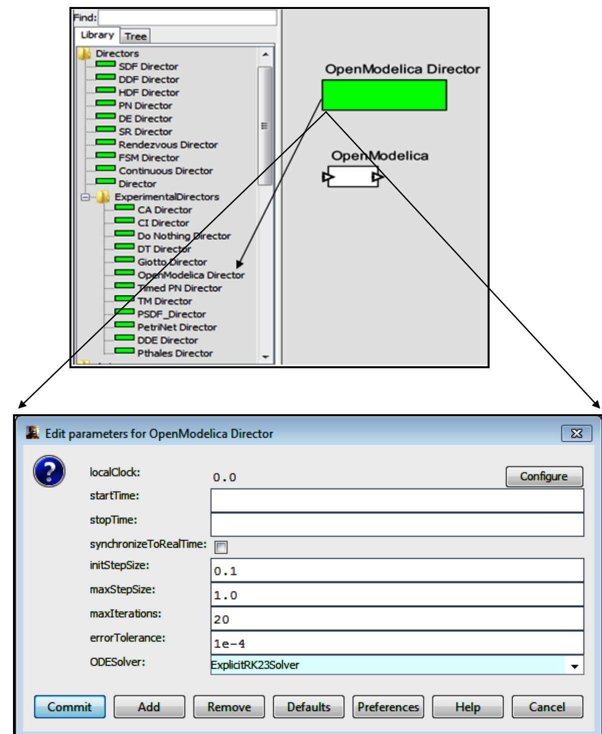

Figure 1: OpenModelicaDirector is provided on the left menu under the Directors → ExperimentalDirectors.

other actors and simulates the Modelica model. This simulation can be done in either batch or interactive processing mode. The results of the simulation can then be passed to Ptolemy II actors, to be displayed or modified.

An execution in Ptolemy II is divided into the following phases: setup, iterate, and wrap-up. The setup phase is divided into two phases, preinitialize and initialize. The preinitialize sub-phase usually handles structural information, such as constructing dynamically created actors, determining the width of ports, and creating receivers. The initialize phase initializes parameters, resets local states, and produces initial tokens. The preinitialization and initialization of an actor are performed exactly once during the actor's life cycle.

To organize the interactions among actors, an iteration is divided into **prefire**, **fire**, and **postfire**.

1. Prefire checks if the preconditions are fulfilled for the actor to execute, such as the presence of sufficient inputs to complete the iteration.

2. Most of the actions are taken place in the fire phase, which involves reading the inputs, processing data, and producing outputs. Some MoCs

DOI 10.3384/ECP140961247

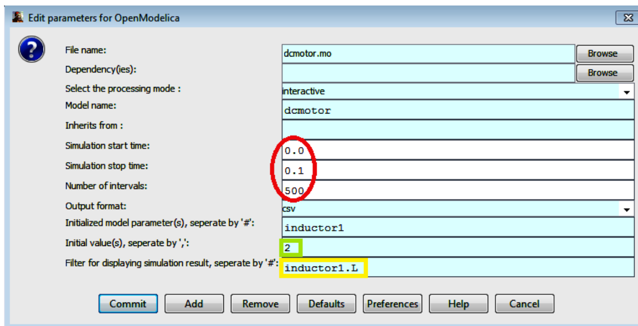

Figure 2: We can see that the model to be loaded is contained in the file *dcmotor.mo*. We can also set the simulation parameters (in red), the parameters of the model(green), such as for instance the resistance of the resistor and apply a filter to select the variables that should be displayed (in yellow).

```
model dcmotor
  Modelica.Electrical.Analog.Basic.Resistor     resistor1(R = 10);
  //Observe the difference between MSL 2.2 and 3.1 regarding the defaul
  Modelica.Electrical.Analog.Basic.Inductor     inductor1(L = 0.2);
  Modelica.Electrical.Analog.Basic.Ground       ground1;
  Modelica.Mechanics.Rotational.Components.Inertia     load(J = 1);
  // Modelica.Mechanics.Rotational.Inertia          load(J = 1); // Mode
  Modelica.Electrical.Analog.Basic.EMF          emf1;
  Modelica.Blocks.Sources.Step                  step1;
  Modelica.Electrical.Analog.Sources.SignalVoltage signalVoltage1;
```

Figure 3: The parameters for the Modelica model can be set through the graphical interface in Vergil.

like synchronous reactive models and CT differential equations support fixed-point iteration which enables the computation of the fixed point of actor outputs while keeping the state of each actor constant.

3. Invocation of `fire()` several times prior to the invocation of `postfire()` results to updating the state of an actor at the time of reaching the fixed point. At the final stage, `wrapup()` is invoked to release resources that were used during execution.

The domain specific OpenModelica actor is defined in the lib sub-package. It is an atomic actor which is visible in Vergil [6], the graphical editor for the Ptolemy II framework.

In this paper, we illustrate the use of the OpenModelica actor to simulate a simple `DCMotor` example. When looking inside the actor, all the simulation parameters are displayed, as illustrated in Figure 2. When this model is parametrized through the Vergil interface, this will initialize the corresponding Modelica model (Figure 3).

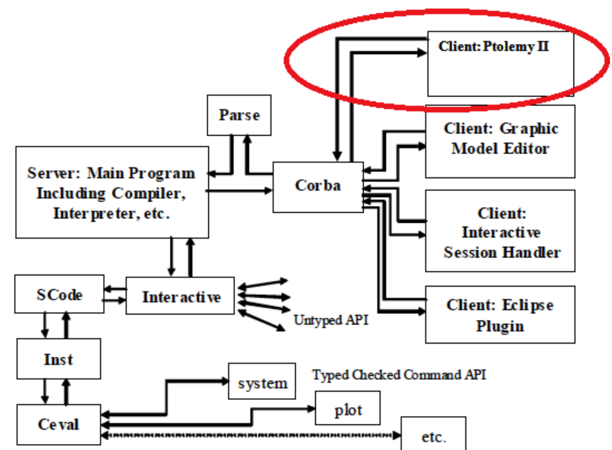

Figure 4: Client-Server interconnection structure of the compiler/interpreter main program and interactive tool interfaces after adding new Ptolemy II as a new client.

3 Simulation

To simulate Modelica models in Ptolemy II, it is necessary to invoke the OpenModelica Compiler (OMC). OMC provides a CORBA interface for remotely invoking the compiler from client applications. This interface is used to communicate with OMC from Ptolemy II (see Figure 4).

An `OpenModelicaDirector` (Figure 1), which extends the ContinuousDirector, is implemented in the Kernel package. The key function of this actor is `postfire()`, a method that will be invoked exactly once during an iteration, after all invocations of the `fire()` method in that iteration. However, in this integration, when `postfire()` is invoked in OpenModelicaDirector it returns false in order to stop/halt the invocation of `fire()` after firing once.

The communication between OpenModelica and Ptolemy follows the following pattern:

1. The OpenModelica actor starts the OpenModelica Compiler(OMC) server in the `initialize()` method, by invoking `startServer()`.

2. In the next step, value(s) of the Modelica parameter(s) are modified by Ptolemy IIs' actor(s) prior to the simulation of the Modelica model through OMC.

3. After simulation of the Modelica model, the retrieved result back from OMC server is plotted/displayed through Ptolemy IIs' actors.

4. Finally, OpenModelica actor sends the result to

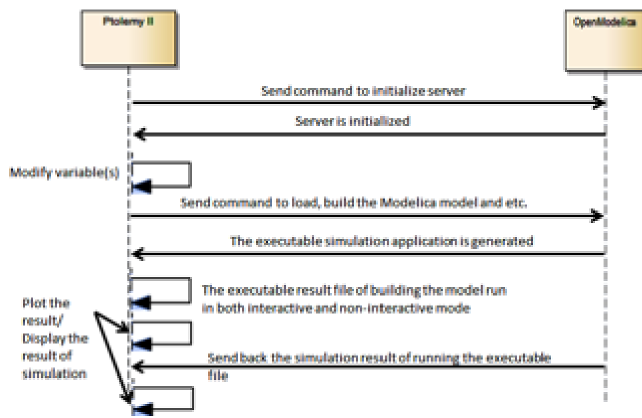

Figure 5: The intercations between Ptolemy II and OpenModelica can be summarized by the following diagram.

the output port which enables using the generated result by other Ptolemy IIs' actors.

All these actions are invoked in the `fire()` method. In the final step, the OMC server is halted through invoking `stopServer()` in in the `wrapup`. These steps are summed up in Figure 5.

The `OMCCommand` class provides the implementation for all the functionalities required by the `initialize()`, `fire()` and `wrapup()` methods, except for modifying the component values after simulation. This functionality is provided by the `UntilSocket` class. Figure 6 presents the class diagram of the implementation, which contains around three thousand lines of code. `OMCCommand` is the largest class, containig over a thousand of code lines. Since the Ptolemy II framework is written in Java, the OpenModelica extensions are also done in Java. Appendix A includes a code snippet that shows the implementation of the `fire()` method.

In order to integrate the OpenModelica actor with components for displating the simulation results in ptolemy, a composite actor is constructed. Figure 7 shows the Composite actor that is used for plotting CSV format, the OpenModelicaDirector that controls the execution order of the OpenModelica actor and the RunCompositeActor, and SDF director which is known as the inside director of RunCompositeActor controls the execution of CSVReader, RecordDisassembler and XYPlotter whenever RunCompositeActor is executed.

In the current implementation both batch and interactive simulation are possible.

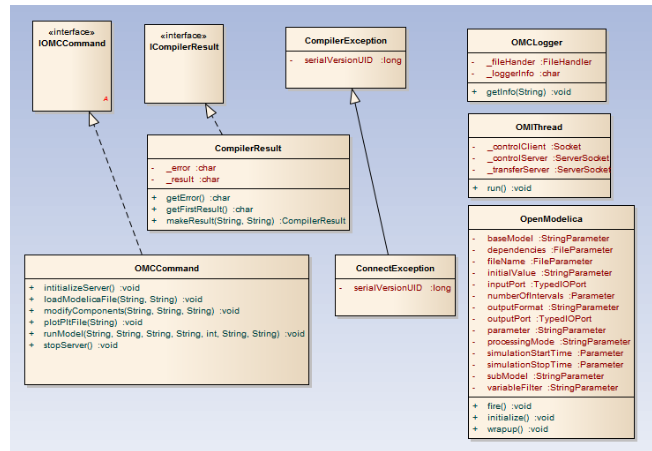

Figure 6: The class diagram for the current implementation.

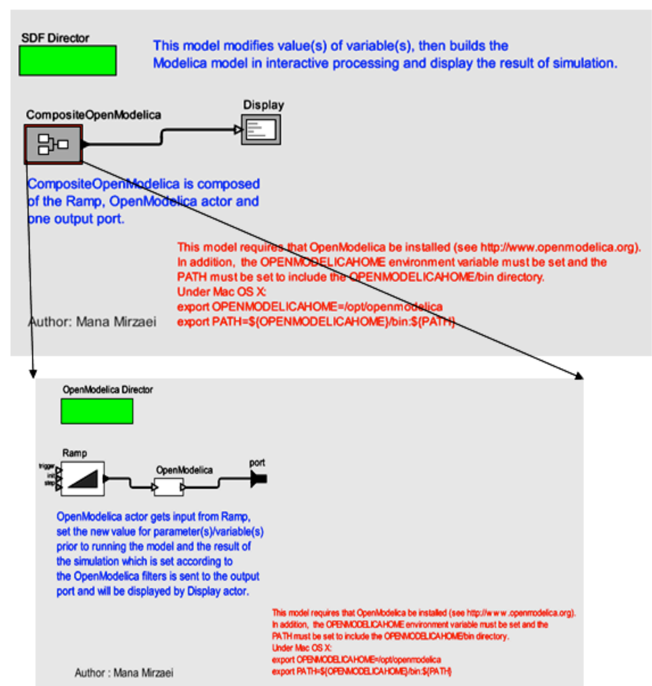

Figure 7: OpenModelicaXYPlotter model is composed of OpenModelica actor and RunCompositeActor.

DOI
10.3384/ECP140961247

3.1 Interactive Simulation

OpenModelica offers a user-interactive and time synchronous simulation known as OpenModelica Interactive (OMI). OMI is part of the simulation runtime core. The output of OMI is an executable simulation application, running the executable file in an interactive processing mode that enables users to govern the simulation runtime behavior.

The integration of OMI with Ptolemy is implemented through the following two modules, illustrated in Figure 8:

Control module is the interface between OMI and Ptolemy II which is implemented as a single thread to support parallel tasks and independent reactivity. The Control module is considered the major controlling and communication instance during the simulation initialization phase as well as for managing simulation properties throughout the simulation runtime. The Control module also reacts to the feedback from other internal OMI components and sends some messages back to Ptolemy II including error and status messages.

Transfer module gets simulation results from a result manager and sends them to Ptolemy II upon launching a simulation. Additionally, the module employs a filter mask allowing the user to select the variables whose result values are significant to Ptolemy II.

At the moment only the step by step simulation of the OpenModelica models is possible, however the next step in the development process is to implement cosimulation. Figure 9 illustrates the results of an interactive simulation, new value(s) for variable(s) of the Modelica model can be set in the *initial value(s)* parameter or the Ramp actor parameter which can be customized in the same way as the OpenModelica actor. The value of the *init* parameter of the Ramp actor overrides the value of *initial value(s)* parameter of OpenModelica actor.

The step time of the simulation in interactive processing is calculated by following formula: *Simulation step time = (Simulation stop time - Simulation start time)/ (Number of intervals)*.

According to the above formula, the step time is 0.0002 [(1.0 - 0.0)/500] in this example. The calculated step time results in the start of the simulation at 0.0002 and completion at 0.0998, as shown in red in Figure 9.

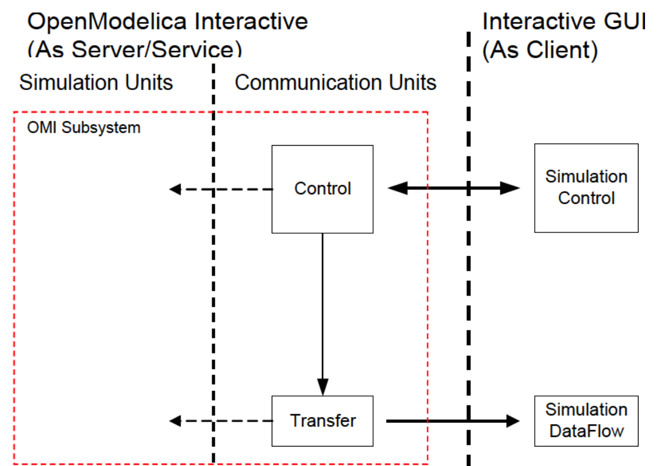

Figure 8: Interactive simulation mechanism.

```
.OpenModelica.Display
File  Help
At time : 0.0002 resistor1.R_actual=2 load.w=2
At time : 0.0004 resistor1.R_actual=2 load.w=2
At time : 0.0006 resistor1.R_actual=2 load.w=2
At time : 0.0008 resistor1.R_actual=2 load.w=2
At time : 0.001 resistor1.R_actual=2 load.w=2
At time : 0.0012 resistor1.R_actual=2 load.w=2
At time : 0.0014 resistor1.R_actual=2 load.w=2
At time : 0.0016 resistor1.R_actual=2 load.w=2
At time : 0.0018 resistor1.R_actual=2 load.w=2
At time : 0.002 resistor1.R_actual=2 load.w=2
At time : 0.0022 resistor1.R_actual=2 load.w=2
At time : 0.0024 resistor1.R_actual=2 load.w=2

At time : 0.0984 resistor1.R_actual=2 load.w=2
At time : 0.0986 resistor1.R_actual=2 load.w=2
At time : 0.0988 resistor1.R_actual=2 load.w=2
At time : 0.0992 resistor1.R_actual=2 load.w=2
At time : 0.0994 resistor1.R_actual=2 load.w=2
At time : 0.0996 resistor1.R_actual=2 load.w=2
At time : 0.0998 resistor1.R_actual=2 load.w=2
```

Figure 9: Displaying part of simulation result of OpenModelicaInteractive.xml by the Display actor.

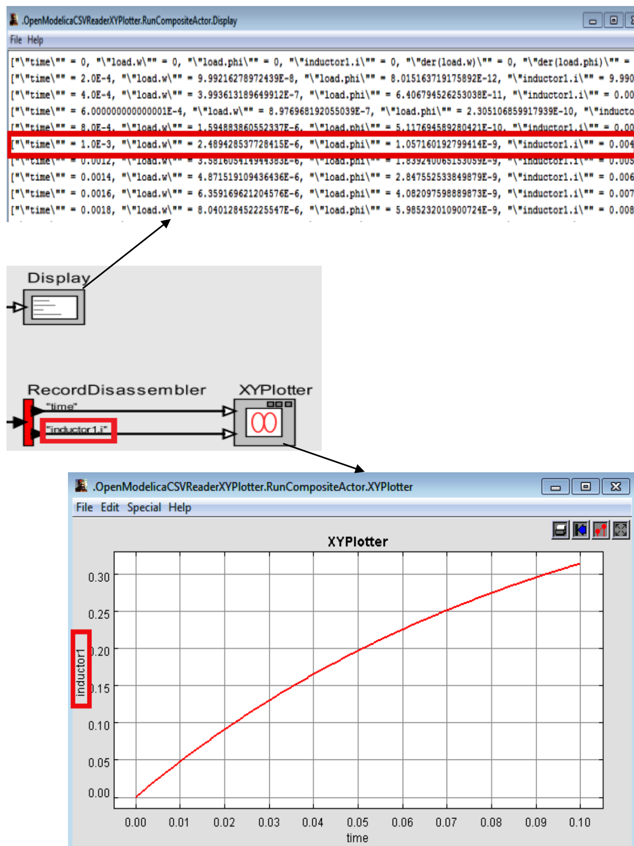

Figure 10: The generated output is displayed in XY-Plotter as well as in textual format by the corresponding actors.

3.2 Non-interactive simulation

There is also the possibility of running the executable simulation application which is generated by OMC in a non-interactive simulation runtime. In contrast to Interactive Simulation, this method does not offer any user-interactive simulation. Running the executable file in batch processing mode allows to generate the whole simulation result in CSV or PLT file formats (Figure 10) that can be displayed and used by other actors in Ptolemy II.

4 Performance

In this section we compare the simulation times of two test models in OpenModelica and when simulating them through Ptolemy, to estimate overhead. The following table compares the two:

Use Case	in Ptolemy II	in OpenModelica
OMXYPlotter	25021ms	30ms
OMPltPlotter	28673ms	12ms

As simulating the models in Ptolemy involves establishing a connection with the OpenModelica compiler and exchanging data, some overhead is to be expected. However it is the tradeoff for simulation in a heterogeneous environment.

5 Related Works

An integration between Dymola, one of the commercial tools for Modelica modeling and simulation, and Ptolemy II through a software environment known as BCVTB4 has been developed [7]. In this approach, Ptolemy II acts as the middleware for implementing BCVTB and actors in Ptolemy II are responsible for starting a server that uses the BSD5 socket utilized for exchanging data between the simulator and the actor as well as implementing the inter-process communication. This approach is implemented by adding BCVTB block to the Dymola library and it is necessary to include this block in the Modelica model in order to enable co-simulation.

FMI (Functional Mockup Interface) is an evolving standard for composing model components designed using distinct modeling tools that can also be used in the cosimulation of models [8].

6 Conclusion

In this article we have presented the integration of Modelica models in the Ptolemy II framework for modeling large-scale heterogeneous concurrent systems. This will allow the simulation of Modelica models through the use of the OpenModelica compiler within the Ptolemy II network.

Moreover the current architecture can serve as a base for further integration efforts, such as the bisimulation of Modelica models in conjunction with other formalisms.

DOI 10.3384/ECP140961247

Appendix A

The following code snippet shows the implementation of the `initialize()` method:

```
/** Invoke the fire() of the super class. Then, Modelica library and model(s) are loaded.
     * Upon modifying the value of variable(s) and parameter(s) by input port or actors'
        parameters,
     * the Modelica model is built in <i>non-interactive</i> or <i>interactive</i> mode.
     * <p>After building the model in an interactive mode, the simulation result
     * is calculated step by step according to the parameters of the OpenModelica actor.
     * The result is sent in the string format to the output port of the OpenModelica
        actor to be
     * displayed by Display actor.</p>
     * @exception IllegalActionException If the evaluation of the expression
     * triggers it, or the evaluation yields a null result, or the evaluation
     * yields an incompatible type, or if there is no director.
     */
 public void fire() throws IllegalActionException {
   super.fire();

   // Load Modelica library and model(s).
   try {
      _omcCommand.loadModelicaFile(fileName.getExpression(),
         subModel.getExpression());
   // If the model is inherited from a base model,
   // that base model should be loaded in advance to the derived model.
   // Otherwise, the derived one could not be built.
     if (!(dependencies.getExpression().isEmpty() && baseModel
          .getExpression().isEmpty()))
      _omcCommand.loadModelicaFile(dependencies.getExpression(),
                     baseModel.getExpression());
     } catch (ConnectException e) {
       throw new IllegalActionException(
           "Unable to load Modelica file/library!" + e.getMessage());
       }

        // There is a value to be passed to the OpenModelica actor's port.
        if (input.getWidth() > 0) {
            // Get the token from input port of OpenModelica actor.
            IntToken inputPort = (IntToken) input.get(0);
            try {
                // Modify components of the Modelica model prior to running the model.
                if (!(parameter.getExpression().isEmpty() && initialValue
                        .getExpression().isEmpty())) {
                    if (!(baseModel.getExpression().isEmpty())) {
                        _omcCommand.modifyComponents(inputPort.toString(),
                                baseModel.getExpression(),
                                parameter.getExpression());
                    } else {
                        _omcCommand.modifyComponents(inputPort.toString(),
                                subModel.getExpression(),
                                parameter.getExpression());
                    }
                } else {
                    _omcLogger
                    .getInfo("There is no component to modify prior to running the model!");
                }
            } catch (ConnectException e) {
```

```
            throw new IllegalActionException(
                    "Unable to modify components' values!" + e.getMessage());
        }
        // There is no value to be passed to the OpenModelica actor's port and the new
            value is set by
        // actors' parameters.
    } else if (!(input.getWidth() > 0)) {
        if (!(parameter.getExpression().isEmpty() && initialValue
                .getExpression().isEmpty())) {
            try {
                if (baseModel.getExpression().isEmpty()) {
                    _omcCommand.modifyComponents(
                            initialValue.getExpression(),
                            subModel.getExpression(),
                            parameter.getExpression());
                } else {
                    _omcCommand.modifyComponents(
                            initialValue.getExpression(),
                            baseModel.getExpression(),
                            parameter.getExpression());
                }
            } catch (ConnectException e) {
                throw new IllegalActionException(
                        "Unable to modify components' values of "
                                + baseModel.getExpression() + " !"
                                + e.getMessage());
            }
        } else {
            _omcLogger
            .getInfo("There is no components to modify prior to running the model!");
        }
    }

    // Build the Modelica model and run the executable result file.
    // Plot the result file of the simulation that is generated in plt format.
    try {
        if (!(dependencies.getExpression().isEmpty() && baseModel
                .getExpression().isEmpty())) {
            _omcCommand.runModel(dependencies.getExpression(),
                    baseModel.getExpression(),
                    simulationStartTime.getExpression(),
                    simulationStopTime.getExpression(),
                    Integer.parseInt(numberOfIntervals.getExpression()),
                    outputFormat.getExpression(),
                    processingMode.getExpression());

            if (outputFormat.getExpression().equalsIgnoreCase("plt")
                    && processingMode.getExpression().equalsIgnoreCase(
                            "non-interactive")) {
                _omcCommand.plotPltFile(baseModel.getExpression());
            }
        } else {
            _omcCommand.runModel(fileName.getExpression(),
                    subModel.getExpression(),
                    simulationStartTime.getExpression(),
                    simulationStopTime.getExpression(),
                    Integer.parseInt(numberOfIntervals.getExpression()),
                    outputFormat.getExpression(),
```


DOI 10.3384/ECP140961247

```
                    processingMode.getExpression());

            if (outputFormat.getExpression().equalsIgnoreCase("plt")
                    && processingMode.getExpression().equalsIgnoreCase(
                            "non-interactive")) {
                _omcCommand.plotPltFile(subModel.getExpression());
            }
        }

        // In case of building the model in an interactive mode, client and servers are
            created,
        // IP and ports of the servers are set and streams for transferring information
            between
        // client and servers are set up all in the constructor of the thread.
        // Through starting the thread, the simulation result is sent from the server to
            the
        // Ptolemy II in the string format.
        if (processingMode.getExpression().equalsIgnoreCase("interactive")) {
            _omiThread = new OMIThread(variableFilter.getExpression(),
                    simulationStopTime.getExpression(), output);
            // FIXME: This method explicitly invokes run() on an object. In general,
                classes implement the Runnable
            // interface because they are going to have their
            // run() method invoked in a new thread, in which case Thread.start() is the
                right method to call.
            _omiThread.run();
        }
    } catch (UnknownHostException e) {
        e.printStackTrace();
        throw new IllegalActionException("Host Exception: "
                + e.getMessage());
    } catch (IOException e) {
        e.printStackTrace();
        throw new IllegalActionException("Socket Connection Error: "
                + e.getMessage());
    } catch (ConnectException e) {
        e.printStackTrace();
        throw new IllegalActionException("ServerError: " + e.getMessage());
    }
}
```

References

[1] Christopher Brooks, Edward A. Lee, Xiaojun Liu, Stephen Neuendorffer, Yang Zhao,Haiyang Zheng, Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II) , April 1, 2008.

[2] Christopher Brooks, Edward A. Lee, Xiaojun Liu, Stephen Neuendorffer, Yang Zhao, Haiyang Zheng, Heterogeneous Concurrent Modeling and Design in Java (Volume 2: Ptolemy II Software Architecture), April 1, 2008.

[3] Peter Fritzson. Principles of Object-Oriented Modeling and Simulation with Modelica 2.1, 940 pages, Wiley-IEEE Press, 2004.

[4] OpenModelica Home Page, www.openmodelica.org, Last Accessed Feb. 2013.

[5] Johan Eker, Jörn W. Janneck, Edward A. Lee, Fellow, Ieee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, And Yuhong Xiong, Taming Heterogeneity-The Ptolemy Approach, Proceedings Of The IEEE, Vol. 91, No. 1, Jan. 2003.

[6] UsingVergil, http://ptolemy.eecs.berkeley.edu/ptolemyII/ptIIlatest/ptII/doc/design/usingVergil/using Vergil.pdf, Last Accessed June 2013.

[7] Michael Wetter, Co-simulation of building energy and control systems with the Building Controls Virtual Test Bed, Journal of Building Performance Simulation, Volume 4, Issue 3, 2011

[8] David Broman, Christopher Brooks, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis and Michael Wetter, Determinate Composition of FMUs for Co-Simulation, EECS Department University of California, Berkeley Technical Report No. UCB/EECS-2013-153 August 18, 2013

DOI 10.3384/ECP140961247